Group Project 2: ReCOP

Ian Kuik, Rishi Shukla
Team #10

Table of Contents

Introduction
Background
Architecture Overview
Fetch and Decode
Execute
    Control Unit
    Regfile
    Arithmetic Logic Unit
    Data Memory
    Registers
Results and Discussion
Conclusion
References

Introduction

In this project, we present a Reactive CoProcessor (ReCOP), comprising of a datapath with a control unit that is aimed to be implemented as a component in a Heterogeneous Multiprocessor System on Chip (HMPSOC), running on a DE2-115 FPGA. Our ReCOP's main objective is to act as an executor of control flow and a processor to establish data flow between application-specific processes, which in our case is a Digital Processing Application.

The following report outlines brief background information on ReCOP, its previous achievements in other projects, an overview of our design and implementation, and an overall discussion of future work.

Background

Embedded systems are known to be reactive, where they respond to events being received at unexpected times and have to be able to handle generating outputs based on these events. Traditional processors often cannot handle/optimise this use case, and whilst the creation of a reactive processor exists, the inadequacies of handling data efficiently and lack of support to compile high-level programming languages prove that there is a gap[1].

However, the idea for a reactive concurrent processor, also known as ReCOP, has emerged, where a reactive processor is made to suit and handle the requirements of reactive applications. Whilst it may have the basic tools to be one, ReCOP is not intended to be a standalone processor and instead is a processor that works alongside traditional processors(Nios, ARM, etc.) to improve the performance and handle reactive behaviours of embedded high-level programming. In contrast, the traditional processor would handle compilation and data-driven events[1].

ReCOP has been beneficial in advancing the idea of hybrid reactive processors and the opportunity these technologies can present, with the most notable achievement of ReCOP being in tangential usage with another processor to execute SystemJ code, a system-level language to allow for embedded programming to be done in the syntax of Java. ReCOP's contribution to this case was a Control Virtual Machine, a custom-built processor that handled all control-driven processes that ran concurrently (hence a reactive co-processor), with a Nios processor that handled data-driven processing.

Overall, due to the architecture of the ReCOP, execution speeds were faster than pure Java implementations of SystemJ and allowed SystemJ to exist as a much more efficient method for embedded programming in comparison to C code and allowed developers to program better with a reduction of bugs, due to not dealing with the convoluted pointer arithmetic and threading implementation seen in C[1,2].

Architecture Overview

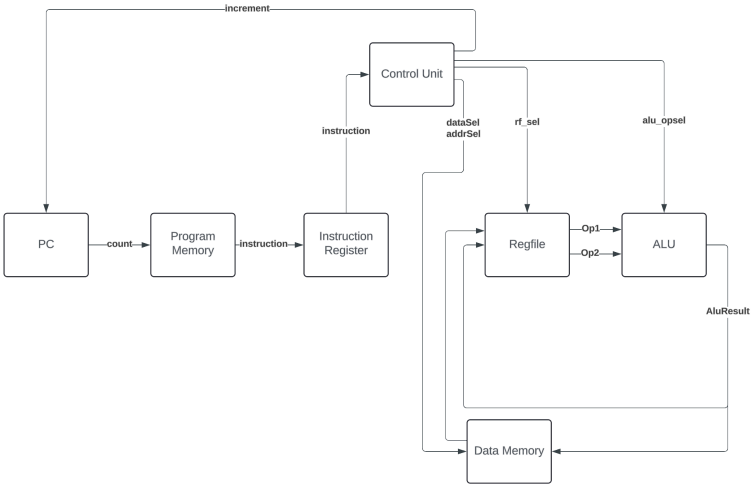

Figure 1: High-Level Diagram of Datapath with Control Unit

Our processor can be broadly classified into the following components:

- **Program Memory(PM)**: Register containing all instructions to be programmed onto our RECOP.
- **Program Counter(PC):** Used to iterate through the program memory's instruction list and enable the deployment of the instruction to the Instruction Register.
- **Instruction Register:** A register that receives 32-bit instructions and decodes by breaking down what the instructions need to do and sending that to the Control Unit while sending any data to the RegFile component.
- **RegFile:** a component which contains registers capable of storing 16-bit data. These store immediate values and short-term information in our various instructions.
- **Arithmetic Logic Unit(ALU):** A component capable of performing Arithmetic operations on our data, such as AND, OR, ADD, SUB and MAX.
- **Data memory**: A data storage unit that stores data in the long term.
- **Control unit:** The 'brain' of this processor outputs control signals to every other component to select values and functionalities that are needed to be performed, allowing the processor to work in a time-critical manner.

Our processor can be reduced to three states: **Fetch, Decode and Execute.**

Fetch and Decode

Our Program Counter (PC) increments through our Program Memory to provide an instruction for our processor to execute.

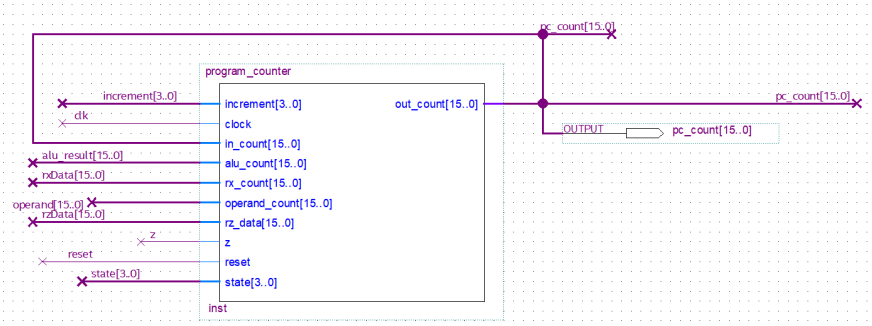

Figure 2: Program Counter block diagram

The most significant input is Increment, a control signal provided by our Control Unit that changes or preserves the output value of the count. The PC will choose to take on the input operand, Rx, and ALU result or implement the SZ instruction through this signal. This output count will index our program memory and retrieve the instruction at that address.

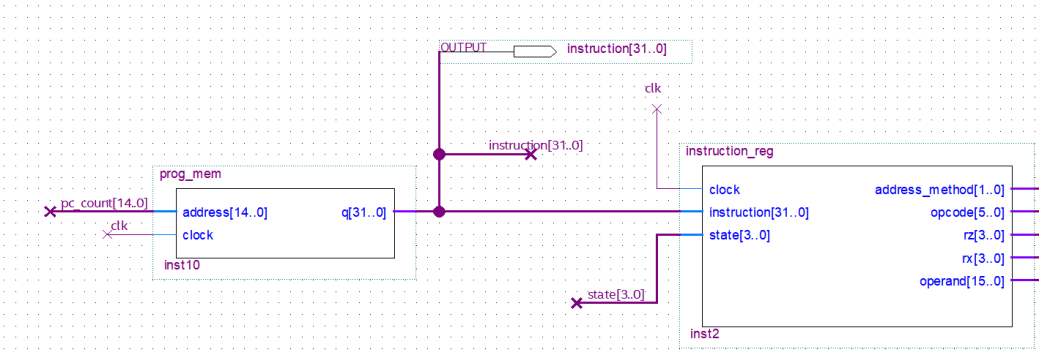

Figure 3: Block diagram of Program Memory and Instruction Register.

Loaded via a Memory Initialization File (MIF) created by our assembler (see compendium), instructions are stored in the program memory of our processor. Since our instructions are 32 bits, each instruction fits into one line, meaning our program counter would only need to increment by one. Since we reduce the number of lines an instruction takes, we have 4096 words, giving us 128kB of Read Only Memory (ROM).

The instructions will be sent into our instruction register at the next clock cycle, decoded as follows, and sent to the control unit for execution. The addressing method is our most significant bits(31 and 30), while Operand is the least significant bits(15 to 0). Our fetch and decode stages are one clock cycle as all components are synchronised to the rising edge of the clock.

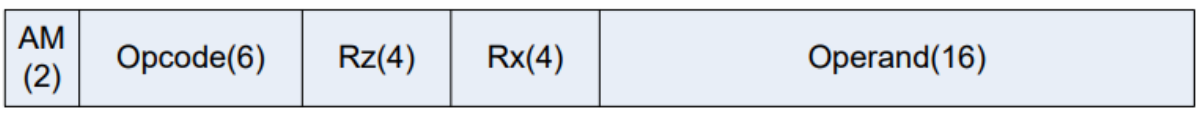

Figure 4: 32-bit instruction sections.

After decoding our instruction and breaking it into their respective parts(shown above), our execution begins with the addressing method and opcode being passed into our control unit.

Execute

Control Unit

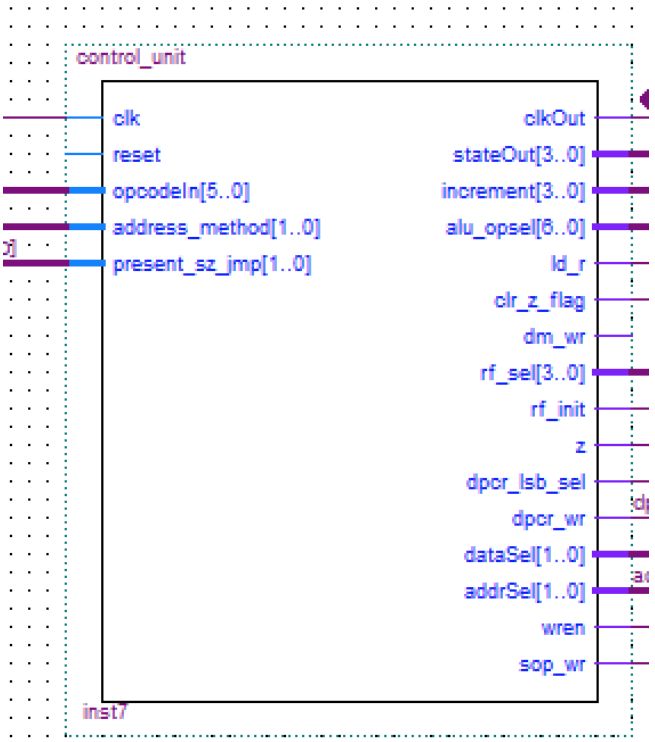

Figure X: Control Unit Block Diagram

The ‘brain’ of this processor has a clock output that ensures all of our components are synchronised, and the state output ensures that our components only perform actions when needed. Increment output changes the Program Counter to fetch a new instruction.

Based on the opcode and addressing method, the following control signals can be outputted:

- **ALU Operand Select:** chooses the operands and operations to be done in the ALU.
- **Load Register(ld_r):** enables data loading into the destination register Rz.
- **Register File Select(rf_sel):** selects the value stored in Rz.
- **Register File Init(rf_init)**: clears all of the registers.
- **Clear Z Flag(z):** when high, clears the Z output of the ALU.
- **DPCR_WR**: enables writing to the output DPCR.
- **DPCR_LSB_SEL:** selects the lower half of DPCR between Rx and Operand.
- **Data Select(dataSel):** selects between Operand, Rz, and Rx to be written into memory.
- **Address Select(addrSel):** selects between Operand, Rz and Rx the address for data to be written to memory
- **Memory Write Enable(wren):** enables the writing of data at the given address in memory
- **Signal Output Write(sop_wr):** enables the writing of data to the signal output

Regfile

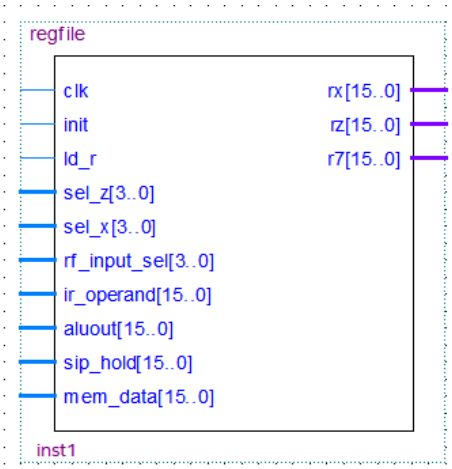

Figure 5: RegFile component block

One of the most essential parts of our execution stage is the reading of the registers Rz and Rx, which are passed on by the instruction. The RegFile component takes in a clock, init, load register (ld_r), the rf_input_sel control signals, and any data from either memory, ALU, Signal Input (SIP) and the operand to be loaded into registers.

Furthermore, our register file component reads the contents of registers if necessary and supplies the content based on the operation sent, such as sending content to the:

- Program Counter to perform the Jump operation.
- Data Memory to perform the Store operations.
- ALU for arithmetic operations.

Arithmetic Logic Unit

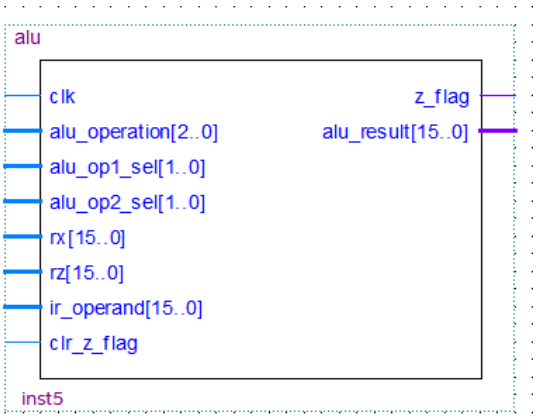

Figure 6: ALU block component

Our ALU handles all of our arithmetic operations by taking in the data provided by RegFile within the specified registers Rz, Rx as well as the operand; it chooses which data to use through the use of the ALU_opsel control signal as well as the operation to be performed.

It can complete AND, OR, ADD, SUB and MAX operations. It outputs the Z flag when the result equals 0, which will be used in SZ operations. Depending on the stated storage method, the output is sent to either the data memory or RegFile.

Data Memory

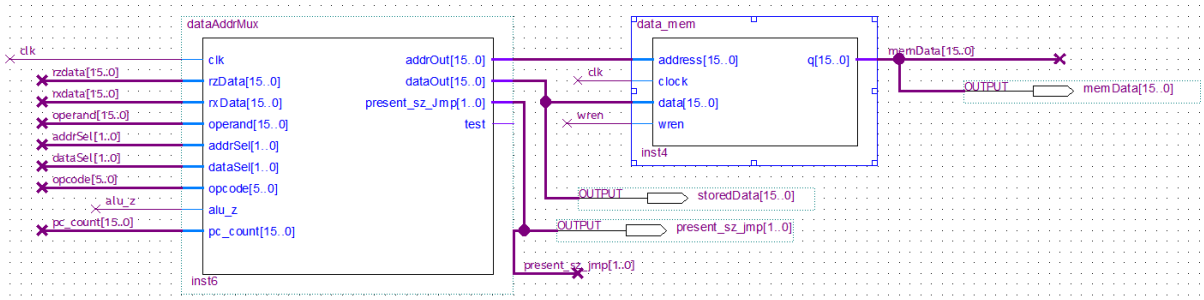

Figure 7: Data Memory and Mux block diagram

Data memory stores data in long-term memory. Preceding the memory component is a data and address mux, which is selected between Rz, Rx, operand, and the current count of PC (for STRPC operations).

Depending on its inputs, dataSel and addrSel will send data and an address to be stored (if write is enabled) or read from memory. Our data memory stores 16-bit data, with one word per line, totalling 64kB of M4K memory.

Registers

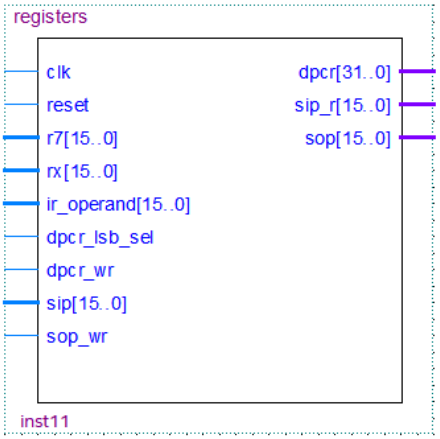

Figure 8: Registers block diagram.

Registers connect our processor to the HMPSOC and enable the reading and writing of Signal Inputs and Outputs (SIP/ SOP). A clock input allows synchronisation with the rest of our processor and control signals present, such as:

- Reset
- sop_wr
- dpcr_lsb_sel
- dpcr_wr

allow us to output our data to other components in our HMPSOC and perform operations on input data. The component inputs include R7, Rx, Operand and a signal input(SIP), where R7, Rx and the Operand are concatenated to act as the DPCR data. At the same time, Rx itself can be outputted through as a signal output(SOP).

Results and Discussion

Our completed ReCOP can perform Signal Outputs (SOP) to the FPGA and take an Input Signal (SIP) from the switches on the FPGA, which was validated by setting the output of our Registers component to the LEDR output ports.

Instruction	**Clock Cycles**
ALU Operations (Add, Or, And, Sub, Max)	7
Load	5 (Immediate) / 6 (Direct/ Register)
Store	6
Jump	3
SZ	4
Present	3
Datacall	5
NOOP	4
SSOP	5
LSIP	5

Table 1: Clock Cycles taken to complete instruction.

A significant improvement can be made to our clock cycles. As seen in the table, there are a few wasted clock cycles due to a waiting period for data to be propagated through components before they can be read. Despite the large clock cycles, our system is time-predictable and sufficient for our applications. However, future work should mitigate large clock cycles to improve the processor's performance.

Another issue we would like to address is the activation of our seven seg displays despite no output being mapped to them. While this did not affect our functionality, this needs to be openly investigated as it may pose a problem if it is decided to implement seven-segment display support.

Furthermore, an issue to look into is the need for a reset functionality on the FPGA, allowing a KEY0 press to reset the Program Counter, DPCR, and SOP. This was not done, as we found that the active low input from the FPGA resulted in constant resets for our ReCOP.

Despite these issues, we have implemented the ReCOP, which effectively carries out all operations stated by the brief and is suitable for use for the HMPSOC if selected for further integration with the systems.

Conclusion

Overall, this project has enabled us to execute the creation of a ReCOP, which will fit into the bigger picture of handling control-driven operations for our Digital Signal Processing applications. Testing shows that the coprocessor is fully integrated with the FPGA, and the instructions are time-predictable and suitable for our future applications. Future work would include mitigating the clock cycles for improved performance, getting a reset button working, and possibly having the lower 20 bits of DPCR outputted on the seven-segment displays, as this could prove meaningful.

References

1. Salcic, Z. S., & Lonergan, H. (2020). ReCOP – A Processor Core for Control-dominated Reactive Applications. The University of Auckland Department of Electrical and Computer Engineering Embedded Systems Research Group.
2. The University of Auckland Department of Electrical and Computer Engineering Embedded Systems Research Group. (2014). ReCOP – Reactive Coprocessor for Tandem Processor Embedded Execution Platform for SystemJ Language.

Task	Time Taken	Delegation of tasks	
		Ian	Rishi
Datapath Design and Implementation	30 Hours		
Control Unit Design and Implementaion	30 Hours		
Control Unit Testing	6 Hours		
Assembler (Creation and Integration)	6 Hours		
FPGA Integration	10 Hours		
Report	10 Hours		